ML and the Address Operator

Michael Sperber
Wilhelm-Schickard-Institut für Informatik
University of Tübingen
72076 Tübingen, Germany
`sperber@informatik.uni-tuebingen.de`

Peter Thiemann
Department of Computer Science
University of Nottingham
University Park
Nottingham, NG7 2RD, England
`pjt@cs.nott.ac.uk`

Abstract

ML supports references to objects through mutable *`ref` cells*: A program can create a `ref` cell from an object, and subsequently replace the object accessible through the cell by another. Unfortunately, `ref` cells, when compared to mechanisms for handling mutable data in other programming languages, impose awkward restrictions on programming style. Moreover, introducing `ref` cells into a program after the fact requires pervasive changes. This paper proposes a language extension to ML with *pointers*, a safe mechanism for dealing with mutable objects. The extension avoids the style and maintenance problems inherent in the `ref` cell mechanism.

Modern programming languages such as Java, Scheme, and ML, use “safety” as an advertisement: A program cannot crash the system by dereferencing an invalid address. This achievement is largely due to two improvements over low-level languages such as C, Modula-2, or Pascal:

Automatic Storage Management Garbage collection obviates the need for `free` or `dispose` calls which are dangerous because they make formerly valid pointers invalid.

References are Boxes A program cannot take the address of an object and write through it later. Instead, it must create a *reference* (or *box*) for mutable data whose contents the program may change and whose extent is unlimited; the object in the box itself still does not become mutable.

This last property appears in different forms across the spectrum—Standard ML has `ref` cells, Objective Caml has record fields which are mutable by declaration, in Java all record fields and bindings are mutable, and Scheme has mutable bindings, pairs and vectors. The expressiveness of all of these mechanisms is essentially the same.

References-as-boxes have a strong bearing on programming pragmatics: While functional programming practice discourages the use of mutable data altogether, it is not always possible to do without it. Moreover, as appealing as the merits of pure functional programming are, mutable data structures *are* a powerful and frequently useful abstraction [1].

In ML, in order for a value to become mutable, the programmer must replace it by a reference—this means changes at all of its occurrences. This is syntactically inconvenient and generally awkward—the mutability of a value is of no consequence for most of its uses. Yet this fact becomes painfully obvious because the value becomes available only by dereference.

For writing new programs, adding references is merely a nuisance. However, the issue becomes more serious when changing old ones, because the ensuing changes are usually far-reaching throughout a program, and the necessary abstractions for delimiting the change are rarely in place already. This is one of the few valid complaints C and C++ programmers have when switching to the safe languages.

In ML, the situation is especially aggravated because ML does not have mutable bindings like the other languages considered. Therefore, the introduction of references is necessary more often here.

We propose an alternative design to the reference mechanism involving a safe version of *pointers*: It is possible to take the pointer of expressions representing locations (so-called *lvalues*), and subsequently modify the contents of the locations themselves. This means that the modifications associated with the introduction of references become completely superfluous. The extension is a generalization of the reference mechanism.

Our extension is obviously a restriction of the address mechanism in C and C++ which allows address arithmetic and address forging by casting. However, in contrast to C and C++, pointers have unlimited extent in our extension. This means, for instance, that a pointer to a local variable may survive its activation record.

Here are contributions of our work:

- We present a safe extension of Standard ML with pointers.
- We define the semantics of the extension by translation into Standard ML.
- We discuss the language design issues resulting from the extension for both Standard ML and Objective Caml.
- We present small but typical examples (an implementation of mutable queues and a Prolog interpreter) which benefit from the extension.
- We discuss implementation issues associated with the extension in the context of a prototype implementation in the Objective Caml system.

$$
\begin{array}{lll}
l & ::= & x \mid \texttt{\#}\ell(e) \mid \texttt{sub(}e\texttt{,}e\texttt{)} \\
e & ::= & c \mid l \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e\ e \\
 & \mid & \texttt{let val}\ x = e\ \texttt{in}\ e\ \texttt{end} \mid \texttt{rec}\ x(x) = e \\
 & \mid & \{\ell_i = e_i\} \mid \texttt{array(}e\texttt{,}e\texttt{)} \\
 & \mid & \texttt{\&}l \mid \texttt{*}e \mid e\ \texttt{*=}\ e
\end{array}
$$

Figure 1: Syntax of expressions

$$
\begin{array}{lll}
\tau & ::= & \alpha \mid \texttt{unit} \mid \texttt{int} \mid \tau \to \tau \\
 & \mid & \{\ell_i : \tau_i\} \mid \tau\ \texttt{array} \mid \tau\ \texttt{ptr} \\
\sigma & ::= & \tau \mid \forall\alpha.\sigma
\end{array}
$$

Figure 2: Syntax of types

1 Extending ML with pointers

In this section, we formally define the syntax of expressions and types for an illustrative subset of ML extended by operations for pointers. Next, we show that the straightforward, naive extension of ML's typing rules by rules for operations on pointers is insufficient. To deal with the problems of the first attempt, we present a revised type system. Finally, we define the semantics of the language using a type-directed translation to Standard ML.

1.1 Expressions and types

A subset of the Standard ML language is the basis of the discussion. The grammar of expressions in Fig. 1 comes in two parts. Some expressions, marked as l in the grammar, are *lvalues* of which a program may take the address. These expressions refer to *locations* in memory during execution. These are variables as well as record and array components. The expressions e are constants c, lvalues l, lambda abstractions, applications, `let` expressions, recursive function declarations, record and array constructors (from the ML Basis Library [24]), and finally the new operations, taking the address of an lvalue `&`l, dereferencing a location `*`e, and assigning through a pointer e `*=` e. The only deviation from standard practice is `rec` which similar to the presentation in Milner's paper [21]. The usual form `let fun` f x `=` e `in` e' `end` is available as `let val` f `= rec` $f(x)$ `=` e `in` e'.

The type language is the same as for Standard ML, extended by the type τ `ptr` representing pointers to values of type τ. Figure 2 summarizes the syntax of types and type schemes.

1.2 A first attempt

A naive attempt at formulating the typing rules looks exactly as in Standard ML. The only addition are the three rules in Fig. 3 for the new constructs dealing with pointers. These rules are quite similar to those dealing with references.

Unfortunately, this approach gives rise to two problems:

- unsoundness in the presence of polymorphism and
- dangling pointers.

Unsoundness The interaction of polymorphism and mutable state is a delicate issue [29]. It is no surprise that adding

$$
\text{(t-addr)}\ \frac{A \vdash l : \tau}{A \vdash \texttt{\&}l : \tau\ \texttt{ptr}}
$$

$$
\text{(t-deref)}\ \frac{A \vdash e : \tau\ \texttt{ptr}}{A \vdash \texttt{*}e : \tau}
$$

$$
\text{(t-assign)}\ \frac{A \vdash e_1 : \tau\ \texttt{ptr} \quad A \vdash e_2 : \tau}{A \vdash e_1\ \texttt{*=}\ e_2 : \texttt{unit}}
$$

Figure 3: Preliminary typing rules

pointers to ML gives rise to the same problems. Indeed, the standard example for the unsoundness in the presence of references converts to an example for the unsoundness in the presence of pointers.

```
let val f = fn z => z in
(&f *= fn y => y + 1;
f true) end
```

Using the preliminary rules, f has type $\forall\alpha.\alpha \to \alpha$ despite its address being taken. Therefore, the type `int` $\to$ `int` of `fn` y `=>` $y+1$ is an instance of f's type and f `true` uses f at instance `bool` $\to$ `bool`. Hence, we expect a run-time error from applying `fn` y `=>` $y+1$ to `true`.

To avoid this unsoundness, the typing rules used in the implementation do not generalize the type of `let`-bound variables whose address is taken. As a result, the translation from ML with pointers to ML with references preserves typability.

Dangling pointers A program can take the address of a locally defined variable and return it out of its scope. For example,

`fn` x `=>` `&`x

is a function that returns the address of a location which contains the value of its parameter. Clearly, the result is a dangling pointer unless the implementation ensures that the addressed location survives the extent of the body of the lambda. Such dangling pointers are a typical source of errors in C programs which must not occur in a safe language. Therefore, we augment the type system by an analysis to detect this situation and have the implementation box the value in these cases. See Sec. 4 for a more detailed discussion of the implementation issues.

1.3 Revised typing rules

To deal with the problems of the naive attempt, we present a revised type system which includes a simple flow analysis to determine the objects whose address is taken and a simple effect analysis which solves the dangling-pointer problem.

Representing region information First, the type language needs to encode more information. For any lvalue, it needs to express

- whether its address is taken and
- whether its address escapes out of its scope.

The four-element lattice $(\mathit{RegionInfo}, \leq)$ shown in Fig. 4 models this information. The important elements in the lattice are *addr* (addressed) and *esc* (escaping). The remaining elements *unboxed* and *boxed* just serve to complete the lattice. Each variable, array type, and field of a record type

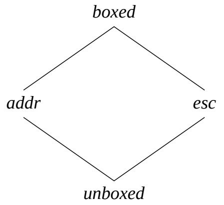

Figure 4: Lattice of address information

$$
\begin{array}{lcl}
\gamma & \in & \mathit{RegionInfoVar} \\
\epsilon & \subseteq & \mathit{RegionInfoVar} \\
\rho & ::= & \alpha \mid \texttt{unit} \mid \texttt{int} \mid \rho \xrightarrow{\epsilon} \rho \\
 & \mid & \{\ell_i : \rho_i(\gamma_i)\} \mid \rho\ \texttt{array}_\gamma \mid \rho\ \texttt{ptr}_\epsilon \\
\pi & ::= & \rho \mid \forall\alpha.\pi
\end{array}
$$

Figure 5: Revised syntax of types

carries a variable $\gamma \in \mathit{RegionInfoVar}$ ranging over the elements of the lattice. If the program takes the address of the respective object, its γ variable must assume a value greater than or equal to *addr*. The typing rules use predicates to describe the possible values of a variable γ. They are relative to an assignment $\Gamma : \mathit{RegionInfoVar} \to \mathit{RegionInfo}$:

- $\mathit{esc}(\gamma)$ iff $\mathit{esc} \leq \Gamma(\gamma)$ and
- $\mathit{addr}(\gamma)$ iff $\mathit{addr} \leq \Gamma(\gamma)$.

The translation uses the predicates to introduce references for all objects whose type has an annotation γ with $\mathit{addr}(\gamma)$. The actual implementation only introduces boxing for objects with $\mathit{addr}(\gamma)$ *and* $\mathit{esc}(\gamma)$.

The type inference algorithm determine a set of constraints on the predicates. Such a set C contains elements of the form $\mathit{addr}(\gamma)$ or $\mathit{esc}(\gamma)$ where $\gamma \in \mathit{RegionInfoVar}$. A solution of such a set C is a map $\Gamma : \mathit{RegionInfoVar} \to \mathit{RegionInfo}$ such that

- for all $\mathit{addr}(\gamma) \in C$, $\mathit{addr} \leq \Gamma(\gamma)$ and
- for all $\mathit{esc}(\gamma) \in C$, $\mathit{esc} \leq \Gamma(\gamma)$.

Since $(\mathit{RegionInfo}, \leq)$ is a complete lattice every set C of constraints has a unique minimal solution.

The pointer type $\rho\ \texttt{ptr}_\epsilon$ carries a set $\epsilon \subseteq \mathit{RegionInfoVar}$ where $\mathit{RegionInfoVar}$ is a set of variables which range over *RegionInfo*. The elements of ϵ belong to the objects which may be addressed by a value of this type. This keeps escaping and non-escaping pointers apart locally.

The function arrow carries the latent effect, that is, the γ values of the pointers that the function may modify. This covers the case that a function takes a local pointer as a parameter and modifies its content.

Figure 5 summarizes the syntax of internal types. None of the annotations present in the type language are visible to the programmer.

Auxiliary functions To state the typing rules, a few more definitions are necessary. The function $\mathit{FA}(\rho)$ defined in Fig. 6 yields the set of *RegionInfoVar* variables contained in the type ρ.

$$
\begin{array}{lcl}
\mathit{FA}(\alpha) & = & \emptyset \\
\mathit{FA}(\texttt{unit}) & = & \emptyset \\
\mathit{FA}(\texttt{int}) & = & \emptyset \\
\mathit{FA}(\rho\ \texttt{ptr}_\epsilon) & = & \epsilon \cup \mathit{FA}(\rho) \\
\mathit{FA}(\rho_2 \xrightarrow{\epsilon} \rho_1) & = & \epsilon \cup \mathit{FA}(\rho_2) \cup \mathit{FA}(\rho_1) \\
\mathit{FA}(\{\ell_i : \rho_i(\gamma_i)\}) & = & \{\gamma_i \mid i\} \cup \bigcup_i \mathit{FA}(\rho_i) \\
\mathit{FA}(\rho\ \texttt{array}_\gamma) & = & \{\gamma\} \cup \mathit{FA}(\rho) \\
\mathit{FA}(\forall\alpha.\pi) & = & \mathit{FA}(\pi)
\end{array}
$$

Figure 6: Collecting *RegionInfoVar*

$$
\begin{array}{lcl}
\mathit{FV}'_C(\alpha) & = & (\{\alpha\}, \emptyset) \\
\mathit{FV}'_C(\texttt{unit}) & = & (\emptyset, \emptyset) \\
\mathit{FV}'_C(\texttt{int}) & = & (\emptyset, \emptyset) \\
\mathit{FV}'_C(\rho\ \texttt{ptr}_\epsilon) & = & \mathit{FV}'_C(\rho) \\
\mathit{FV}'_C(\rho_2 \xrightarrow{\epsilon} \rho_1) & = & (\mathit{FV}(\rho_2) \cup \mathit{FV}(\rho_1), \emptyset) \\
\mathit{FV}'_C(\{\ell_i : \rho_i(\gamma_i)\}) & & \\
 & = & \mathit{let}\ (F_i, G_i) = \mathit{FV}'_C(\rho_i) \\
 & & \quad G'_i = \begin{cases} F_i \cup G_i & \mathit{addr}(\gamma_i) \in C \\ G_i & \text{otherwise} \end{cases} \\
 & & \mathit{in}\ (\bigcup_i F_i, \bigcup_i G'_i) \\
\mathit{FV}'_C(\rho\ \texttt{array}_\gamma) & = & \mathit{let}\ (F, G) = \mathit{FV}'_C(\rho) \\
 & & \quad G' = \begin{cases} F \cup G & \mathit{addr}(\gamma) \in C \\ G & \text{otherwise} \end{cases} \\
 & & \mathit{in}\ (F, G') \\
 & & \\
\mathit{FV}(\rho) & = & \mathit{let}\ (F, G) = \mathit{FV}'_{\{\}}(\rho)\ \mathit{in}\ F \\
\mathit{FV}(\forall\alpha.\pi) & = & \mathit{FV}(\pi) \setminus \alpha
\end{array}
$$

Figure 7: Free variables

ρ is a generic instance of π, $\pi \leq \rho$, iff $\pi = \forall\alpha_1 \ldots \alpha_n.\rho_0$ and $\rho_1, \ldots, \rho_n$ exist such that $\rho = \rho_0[\alpha_i \mapsto \rho_i]$.

The *GEN* function used in the (let) rule that usually maps an environment and a type to a type scheme needs to be adapted. The revised version takes an additional parameter γ which is the *RegionInfoVar* of the `let`-bound variable. If $\mathit{addr}(\gamma)$ or if the type is an array or record type annotated with γ such that $\mathit{addr}(\gamma)$ (and so on recursively) then the type is not generalized.

The new *GEN* function needs to extract the free type variables from a type, except those that occur free inside of an array or record type with a *RegionInfoVar* where $\mathit{addr}(\gamma)$. This function $\mathit{FV}'_C(\rho)$ returns a pair whose first component is the set of all free variables; the second component is the set of those free variables that occur as indicated above. Both functions depend on the constraint set C. Figure 7 shows its definition as well as that of $\mathit{FV}(\pi)$ which extracts all free variables out of types and type schemes.

If $\mathit{FV}'_C(\rho) = (V_1, V_2)$ then the GEN function abstracts over the variables in $V_1 \setminus V_2$.

$$
\mathit{GEN}(C, A, \rho, \gamma) = \begin{cases}
\rho & \mathit{addr}(\gamma) \\
\forall\alpha_1 \ldots \alpha_n.\rho & \text{otherwise} \\
\qquad\qquad \text{where} & \\
\qquad\qquad (F, G) \ = & \mathit{FV}'_C(\rho) \\
\qquad \{\alpha_1, \ldots, \alpha_n\} \ = & (F \setminus G) \setminus \mathit{FV}(A)
\end{cases}
$$

$$\text{(l-var)}\ \frac{x : \rho(\gamma) \in A}{C, A \vdash_l x : \rho/\gamma, \emptyset}$$

$$\text{(l-record)}\ \frac{C, A \vdash_e e : \{\ldots \ell : \rho(\gamma) \ldots\}, \epsilon}{C, A \vdash_l \texttt{\#}\ell\texttt{(}e\texttt{)} : \rho/\gamma, \epsilon}$$

$$\text{(l-array)}\ \frac{C, A \vdash_e e_1 : \rho\ \texttt{array}_\gamma, \epsilon_0 \quad C, A \vdash_e e_2 : \texttt{int}, \epsilon_1}{C, A \vdash_l \texttt{sub(}e_1\texttt{,}e_2\texttt{)} : \rho/\gamma, \epsilon_0 \cup \epsilon_1}$$

Figure 8: Typing rules for lvalues

$$\text{(addr)}\ \frac{C, A \vdash_l l : \rho/\gamma, \epsilon_0 \quad \gamma \in \epsilon \quad addr(\gamma) \in C}{C, A \vdash_e \texttt{\&}l : \rho\ \texttt{ptr}_\epsilon, \epsilon_0}$$

$$\text{(deref)}\ \frac{C, A \vdash_e e : \rho\ \texttt{ptr}_\epsilon, \epsilon_1}{C, A \vdash_e \texttt{*}e : \rho, \epsilon \cup \epsilon_1}$$

$$\text{(assign)}\ \frac{C, A \vdash_e e_1 : \rho\ \texttt{ptr}_\epsilon, \epsilon_1 \quad C, A \vdash_e e_2 : \rho, \epsilon_2}{C, A \vdash_e e_1\ \texttt{*=}\ e_2 : \texttt{unit}, \epsilon \cup \epsilon_1 \cup \epsilon_2}$$

Figure 9: Typing rules for operations involving pointers

Typing rules The typing rules for the internal type language define a simple effect system. It is essentially an effect-monomorphic version of the one by Lucassen and Gifford [19]. Here, a type environment is a finite map that associates a variable x to a pair of a type scheme π and a *RegionInfoVar* γ, indicating the addressing property of the variable. This association is written $x : \rho(\gamma)$.

There are two judgements

- for lvalues: $C, A \vdash_l l : \rho/\gamma, \epsilon$ means that with constraints C and in type environment A the lvalue l has type ρ, *RegionInfoVar* γ, and effect ϵ;
- for expressions: $C, A \vdash_e e : \rho, \epsilon$ means that with constraints C and in type environment A the expression e has type ρ and effect ϵ.

Figure 8 presents the typing rules for lvalues. In the variable rule (l-var), the consequence of the rule is taken from the type assumption so that γ reflects the *RegionInfoVar* value of the variable itself.

The next set of rules—shown in Fig. 9—deals with the operations involving pointers. In rule (addr), if an lvalue has type ρ and *RegionInfoVar* γ then C must contain $addr(\gamma)$ so that every solution Γ of C obeys $addr \leq \Gamma(\gamma)$. Furthermore, the type of the pointer is $\rho\ \texttt{ptr}_\epsilon$ for some ϵ which γ. The reason for having a *set* of *RegionInfoVar* ϵ is that some of the $\gamma \in \epsilon$ may be escaping while others are not.

The remaining rules in Fig. 10 deal with the other expressions. The rule (abs) for lambda abstractions needs to check whether the address of the abstracted variable x escapes from the scope of the function. It does this in the style of the effect-masking rule of an effect system [19]: if the region of the pointer is neither mentioned in the type assumption nor in the result type then the effect on this region is purely local to the body of the function and can be removed from the resulting effect. Our rule does not remove the effect, it only enforces $esc(\gamma)$ for x if γ is mentioned in the type assumption or in the type of the result.

The rule (app) for application is standard for effect systems. The rule (let) employs the same logic as the (abs) rule

$$\text{(var)}\ \frac{x : \pi(\gamma) \in A \quad \pi \leq \rho}{C, A \vdash_e x : \rho, \emptyset}$$

$$\text{(const)}\ C, A \vdash_e c : \rho_c, \emptyset$$

$$\text{(abs)}\ \frac{\begin{array}{c} C, A\{x : \rho_2(\gamma_2)\} \vdash_e e : \rho_1, \epsilon \\ \gamma_2 \in FA(A) \cup FA(\rho_2) \cup FA(\rho_1) \Rightarrow esc(\gamma_2) \in C \end{array}}{C, A \vdash_e \texttt{fn}\ x\ \texttt{=>}\ e : \rho_2 \xrightarrow{\epsilon} \rho_1, \emptyset}$$

$$\text{(app)}\ \frac{C, A \vdash_e e_1 : \rho_2 \xrightarrow{\epsilon} \rho_1, \epsilon_1 \quad C, A \vdash_e e_2 : \rho_2, \epsilon_2}{C, A \vdash_e e_1\ e_2 : \rho_1, \epsilon \cup \epsilon_1 \cup \epsilon_2}$$

$$\text{(let)}\ \frac{\begin{array}{c} C, A \vdash_e e_1 : \rho_1, \epsilon_1 \\ C, A\{x : GEN(C, A, \rho_1, \gamma)\} \vdash_e e_2 : \rho_2, \epsilon_2 \\ \gamma \in FA(A) \cup FA(\rho_2) \Rightarrow esc(\gamma) \in C \end{array}}{C, A \vdash_e \texttt{let val}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \rho_2, \epsilon_1 \cup \epsilon_2}$$

$$\text{(rec)}\ \frac{\begin{array}{c} C, A\{f : \rho_2 \xrightarrow{\epsilon} \rho_1(\gamma_1)\}\{x : \rho_2(\gamma_2)\} \vdash_e e : \rho_1, \epsilon \\ \gamma_2 \in FA(A) \cup FA(\rho_2) \cup FA(\rho_1) \Rightarrow esc(\gamma_2) \in C \end{array}}{C, A \vdash_e \texttt{rec}\ f(x) = e : \rho_2 \xrightarrow{\epsilon} \rho_1, \emptyset}$$

$$\text{(sub)}\ \frac{C, A \vdash_e e : \rho, \epsilon}{C, A \vdash_e e : \rho, \epsilon'} \quad \epsilon \subseteq \epsilon'$$

$$\text{(e-record)}\ \frac{C, A \vdash_e e : \{\ldots \ell : \rho(\gamma) \ldots\}, \epsilon_0}{C, A \vdash_e \texttt{\#}\ell\texttt{(}e\texttt{)} : \rho, \epsilon_0}$$

$$\text{(i-record)}\ \frac{C, A \vdash_e e_i : \rho_i, \epsilon_i}{C, A \vdash_e \{\ell_i = e_i\} : \{\ell_i : \rho_i(\gamma_i)\}, \bigcup \epsilon_i}$$

$$\text{(e-array)}\ \frac{C, A \vdash_e e_1 : \rho\ \texttt{array}_\gamma, \epsilon_0 \quad C, A \vdash_e e_2 : \texttt{int}, \epsilon_1}{C, A \vdash_e \texttt{sub(}e_1\texttt{,}e_2\texttt{)} : \rho, \epsilon_0 \cup \epsilon_1}$$

$$\text{(i-array)}\ \frac{C, A \vdash_e e_1 : \texttt{int}, \epsilon_1 \quad C, A \vdash_e e_2 : \rho, \epsilon_2}{C, A \vdash_e \texttt{array(}e_1\texttt{,}e_2\texttt{)} : \rho\ \texttt{array}_\gamma, \epsilon_1 \cup \epsilon_2}$$

Figure 10: Typing rules for expressions

for detecting whether the pointer to the variable x escapes from the scope of the let. The (rec) rule for defining a recursive function is similar. The (sub) rule for subeffecting is standard in effect systems.

Finally, there are the introduction and elimination rules for records and arrays. As already mentioned, each field of a record has its own region value whereas an array has only one region value. These values are created in the respective introduction rules (i-record) and (i-array). The elimination rules (e-record) and (e-array) just discard them.

Type inference unifies all $\gamma \in$ *RegionInfoVar* for record and array types. However, it does not unify the elements of $\epsilon \subseteq$ *RegionInfoVar* attached to the pointer type constructor and with the function type constructor but collects them in a set, using subeffecting as indicated in rules (sub) and (addr).

Since the typing rules are syntax-directed, the result type ρ in $C, A \vdash_e e : \rho, \epsilon$ is determined by e up to the annotation with $\gamma \in$ *RegionInfoVar*. However, the constraint set C is critical. It is possible to break typability by including new constraints in C as well as by removing constraints. For example, consider

$$\begin{array}{ll} e \equiv & \texttt{let val}\ x = 1\ \texttt{in} \\ & \texttt{let val}\ f = \texttt{fn}\ z\ \texttt{=>}\ z\ \texttt{in} \\ & f\ \texttt{(*(}f\ \texttt{\&}x\texttt{))}\ \texttt{end end} \end{array}$$

$$
\begin{array}{lcl}
|\alpha|^\Gamma & \equiv & \alpha \\
|\texttt{unit}|^\Gamma & \equiv & \texttt{unit} \\
|\texttt{int}|^\Gamma & \equiv & \texttt{int} \\
|\rho\ \texttt{ptr}_\epsilon|^\Gamma & \equiv & |\rho|\ \texttt{ref} \\
|\rho_2 \xrightarrow{\epsilon} \rho_1|^\Gamma & \equiv & |\rho_2|^\Gamma \to |\rho_1|^\Gamma \\
|\rho\ \texttt{array}_\gamma|^\Gamma & \equiv & \begin{cases} |\rho|^\Gamma\ \texttt{ref array} & addr(\gamma) \\ |\rho|^\Gamma\ \texttt{array} & \text{otherwise} \end{cases} \\
|\{\ell_i : \rho_i(\gamma_i)\}|^\Gamma & \equiv & \{\ell_i : \rho_i'\} \\
& & \text{where } \rho_i' \equiv \begin{cases} |\rho_i|^\Gamma\ \texttt{ref}\ addr(\gamma_i) \\ |\rho_i|^\Gamma \quad \text{otherwise} \end{cases} \\
|\forall\alpha.\sigma|^\Gamma & \equiv & \forall\alpha.|\sigma|^\Gamma
\end{array}
$$

Figure 11: Translation of types

$$
\begin{array}{lcl}
|x|_l^\Gamma & \equiv & x \\
|\texttt{\#}\ell\texttt{(}e\texttt{)}|_l^\Gamma & \equiv & \texttt{\#}\ell\texttt{(}|e|_e^\Gamma\texttt{)} \\
|\texttt{sub(}e_1\texttt{,}e_2\texttt{)}|_l^\Gamma & \equiv & \texttt{sub(}|e_1|_e^\Gamma\texttt{,}|e_2|_e^\Gamma\texttt{)}
\end{array}
$$

Figure 12: Translation of lvalues

and suppose x and f are associated with γ_x and γ_f, respectively. Obviously, $C, \{\} \vdash_e e : \texttt{int}, \{\gamma_x\}$ with $C = \{addr(\gamma_x)\}$ but it typechecks neither with $C = \{\}$ nor with $C = \{addr(\gamma_x), addr(\gamma_f)\}$. In general, all we can say is:

Lemma 1 *If $C, A \vdash_e e : \rho, \epsilon$ then there is a unique minimal (with respect to set inclusion) $C' \subseteq C$ such that $C', A \vdash_e e : \rho, \epsilon$.*

Implementation issues A type checker must determine the unique minimal set of constraints. It extends the well-known algorithm $\mathcal{W}$ [21] by constraint handling. The implementation of let polymorphism becomes more complicated. Due to the top-down nature of $\mathcal{W}$ it may be impossible to determine $FV'()$ at the place where it is needed. Therefore, the implementation provisionally employs the standard definition of *GEN* and keeps track of all uses of let-bound variables. Whenever the address of a let-bound variable is taken, the algorithm unifies the instantiated types at all uses of the variable with the type before generalization.

1.4 Semantics

The dynamic semantics is defined by translation to Standard ML. The translation introduces `ref`s for all objects with a γ such that $addr(\gamma)$. Strictly speaking, the translation maps a type derivation in the previously explained system to an ML term.

There are three translation functions, $|.|_e^\Gamma$ for expressions, $|.|_l^\Gamma$ for lvalues, and $|.|^\Gamma$ for types. All translations take a parameter Γ, which is a solution of the constraint set C derived by the typing judgement. Figure 11 shows the type translation. A type assumption A translates to $|A|^\Gamma$ where for each $x : \rho(\gamma)$ in A there is either $x : |\rho|^\Gamma$ `ref` in $|A|^\Gamma$ if $addr(\gamma)$ or $x : |\rho|^\Gamma$ in $|A|^\Gamma$, otherwise.

Figure 12 presents the translation for lvalues. It applies only to addressed objects; the type of the transformed expression always has type τ `ref` for some τ.

Figure 13 defines the translation for expressions. The translation of `rec` deserves some explanation: the tricky issue is that the program may take the address of the function itself. In this case, the translation first builds a `ref` cell which contains a function of the right type but which the program never applies. Next, it updates the `ref` cell with the actual translation, using the update to implement the fixpoint. The translation of arrays needs to take into account that there may be a pointer to each individual element of the array. Therefore, each element must contain a different `ref` cell. The translation does not rely on the mutability of Standard ML's arrays. The remaining constructs are straightforward.

The following theorem states that the translation preserves typing so that the semantics of our language is well-defined via this translation.

Theorem 1 *Suppose $C, A \vdash_e e : \rho, \epsilon$ and let Γ be a solution of C then $|A|^\Gamma \vdash_{ML} |e|_e^\Gamma : |\rho|^\Gamma$.*

1.5 Polymorphic equality

The semantics of polymorphic equality in ML is not quite compatible with the translation: In ML, equality compares the addresses of `ref` cells but the contents of everything else. Thus, the automatic introduction of `ref` cells affects equality tests in the program. Hence, polymorphic equality must distinguish between "transparent" `ref` cells and those at pointer types. Equality must compare the former by content and the latter by address value. This is easily enforced by having the translation expand equality, too. At the implementation level, this problem is straightforward to solve; see Sec. 4.5.

2 Language design issues

In the real world, the base language for the pointer extension is Standard ML or Objective Caml (Caml for short from now on). The extension is straightforward and unproblematic in both cases, but the syntactic issues involved in adding the extensions to these two dialects differ slightly.

ML has two additional aggregate data types not covered in the formal treatment: tuples and elements of algebraic data types. Naturally, it is desirable to also classify the components of these aggregates as lvalues. In Standard ML, tuples are not really a separate issue as they are special cases of records. In Caml, tuples need special treatment.

Here are the pragmatic issues in formulating the syntax of the extension:

2.1 Lvalue notation

All lvalues need to have a "direct" notation rather than an indirect one via pattern matching. Pattern matching creates bindings and therefore new locations; taking pointers of the newly bound variables does not serve the intended purpose. Both Standard ML and Caml lack such notation for the argument of a constructed object. Caml lacks notation for the component of a tuple.

In Standard ML, a program can retrieve the argument to a constructor c from the value of an expression e via the new syntax

`#`c`(`e`)`

which is equivalent to

$$
\begin{array}{lcl}
|c|_e^\Gamma & \equiv & c \\
|x:\rho(\gamma)|_e^\Gamma & \equiv & \begin{cases} \texttt{!}x & \text{if } \mathit{addr}(\gamma) \\ x & \text{otherwise} \end{cases} \\
|\texttt{\#}\ell(e) : \{\ldots \ell : \rho(\gamma) \ldots\}|_e^\Gamma & \equiv & \begin{cases} \texttt{!\#}\ell(|e|_e^\Gamma) & \text{if } \mathit{addr}(\gamma) \\ \texttt{\#}\ell(e) & \text{otherwise} \end{cases} \\
|\texttt{sub(}e_1\texttt{,}e_2\texttt{)} : \rho\ \texttt{array}_\gamma|_e^\Gamma & \equiv & \begin{cases} \texttt{!sub(}|e_1|_e^\Gamma\texttt{,}|e_2|_e^\Gamma\texttt{)} & \text{if } \mathit{addr}(\gamma) \\ \texttt{sub(}|e_1|_e^\Gamma\texttt{,}|e_2|_e^\Gamma\texttt{)} & \text{otherwise} \end{cases} \\
|\texttt{fn}\ x : \rho(\gamma)\ \texttt{=>}\ e|_e^\Gamma & \equiv & \begin{cases} \texttt{fn}\ x\ \texttt{=> let val}\ x = \texttt{ref}\ x\ \texttt{in}\ |e|_e^\Gamma\ \texttt{end} \\ \qquad \text{if } \mathit{addr}(\gamma) \\ \texttt{fn}\ x\ \texttt{=>}\ |e|_e^\Gamma \\ \qquad \text{otherwise} \end{cases} \\
|e_1\ e_2|_e^\Gamma & \equiv & |e_1|_e^\Gamma\ |e_2|_e^\Gamma \\
|\texttt{let val}\ x : \rho(\gamma) = e_1\ \texttt{in}\ e_2\ \texttt{end}|_e^\Gamma & \equiv & \begin{cases} \texttt{let val}\ x = \texttt{ref}\ |e_1|_e^\Gamma\ \texttt{in}\ |e_2|_e^\Gamma\ \texttt{end} \\ \qquad \text{if } \mathit{addr}(\gamma) \\ \texttt{let val}\ x = |e_1|_e^\Gamma\ \texttt{in}\ |e_2|_e^\Gamma\ \texttt{end} \\ \qquad \text{otherwise} \end{cases} \\
|\texttt{rec}\ f : \rho_1(\gamma_1)\texttt{(}x : \rho_2(\gamma_2)\texttt{)} = e|_e^\Gamma & \equiv & \begin{cases} \texttt{rec}\ f\texttt{(}x\texttt{)} = e' \\ \qquad \text{if } \neg\mathit{addr}(\gamma_1) \wedge \neg\mathit{addr}(\gamma_2) \\ \texttt{rec}\ f\texttt{(}x\texttt{)} = \texttt{let val}\ x = \texttt{ref}\ x\ \texttt{in}\ e'\ \texttt{end} \\ \qquad \text{if } \neg\mathit{addr}(\gamma_1) \wedge \mathit{addr}(\gamma_2) \\ \texttt{let val}\ f = \texttt{ref (rec}\ f\texttt{(}x\texttt{)} = e''\texttt{) in} \\ \quad \texttt{let val}\ z = f\ \texttt{:=}\ |\texttt{fn}\ x : \rho_2(\gamma_2)\ \texttt{=>}\ e|_e^\Gamma\ \texttt{in} \\ \quad \texttt{!}f\ \texttt{end end} \\ \qquad \text{otherwise} \\ \qquad \text{where } e' = |e|_e^\Gamma \\ \qquad \text{and } e'' = e'[f \mapsto \texttt{ref}\ f] \end{cases} \\
|\{\ell_i : \rho_i(\gamma_i) = e_i\}|_e^\Gamma & \equiv & \{\ell_i = e_i'\} \\
& & \text{where } e_i' = \begin{cases} \texttt{ref}\ |e_i|_e^\Gamma & \text{if } \mathit{addr}(\gamma_i) \\ |e_i|_e^\Gamma & \text{otherwise} \end{cases} \\
|\texttt{array(}e_1\texttt{,}e_2\texttt{)} : \rho\ \texttt{array}_\gamma|_e^\Gamma & \equiv & \begin{cases} \texttt{let val}\ z = |e_2|_e^\Gamma\ \texttt{in} \\ \quad \texttt{tabulate(}|e_1|_e^\Gamma\texttt{,(fn}\ z'\ \texttt{=> ref}\ z\texttt{)) end} \\ \qquad \text{if } \mathit{addr}(\gamma) \\ \texttt{array(}|e_1|_e^\Gamma\texttt{,}|e_2|_e^\Gamma\texttt{)} \\ \qquad \text{otherwise} \end{cases} \\
|\texttt{\&}l|_e^\Gamma & \equiv & |l|_l^\Gamma \\
|\texttt{*}e|_e^\Gamma & \equiv & \texttt{!}|e|_e^\Gamma \\
|e_1\ \texttt{*=}\ e_2|_e^\Gamma & \equiv & |e_1|_e^\Gamma\ \texttt{:=}\ |e_2|_e^\Gamma
\end{array}
$$

Figure 13: Translation of expressions

`case` e `of` c `x -> x`

(Disambiguation between record field access and constructor argument access is possible by the type of the argument.)

Constructor argument access raises a pattern matching failure if the accessed object does not have the correct constructor.

In Caml, the new syntax

e`.<`*int*`>`

is responsible for tuple component access; this blends well with Caml's already existing constructs for record and array component access.

Similarly, since Caml uses dot notation for record component access, it is natural to use a similar notation to access a constructor argument. Hence,

e`.`c

does the trick. Unlike Standard ML, Caml does not allow the translation of this expression into ordinary Caml if c has a tuple type; in that case, the expression

`match` e `with` c `x -> x`

is not syntactically correct, and the compiler rejects it.

All of the new constructs build lvalues. Their integration is straightforward and does not introduce any new difficulties.

2.2 Simulating references

Both `ref` cells in Standard ML as well as the `mutable` declaration for record components in Caml become superfluous.

A one-component record can model a reference; the definitions of dereference and assignment follow easily:

```
type 'a ref = { val : 'a }
fun ref x = { val = x }
fun ! { val = x } = x
fun r := x = &(#val r) *= x
```

Of course, a substantially more idiomatic translation is:

```
type 'a ref = 'a pointer
fun ref x = &x
fun ! x = * x
fun r := x = r *= x
```

However, the latter version requires *two* heap objects for representing a `ref` cell (the pointer itself and the object holding the location referencing the object itself), whereas the one using a record requires only one just as before—the record itself.

In Caml, `mutable` goes away, and the following translation takes the place of record component assignment:

$x.l$ `<-` $e \Longrightarrow$ `&(`$x.l$`) *=` e

2.3 Mutability declarations

Our framework automatically infers mutability information. It is also feasible to require explicit declaration of mutable values. This would make the framework slightly more conservative as it would introduce monovariance. In our system, each occurrence of a record type may have a different mutability property. Moreover, since the type system still needs to infer representation information, it is not clear what the gain of this requirement would be. The spirit of ML is to infer such information automatically and report it to the user.

The place where explicit declarations are relevant is in module interfaces. A `mutable` qualifier can mark mutable values in declarations.

3 Example application

Mutable queues are a good example for the programming style of ML with pointer. Here is the Standard ML signature of a rudimentary implementation of queues:

```
type 'a queue
exception Queue_empty;
val newq   : unit -> 'a queue
val mtq    : 'a queue -> bool
val addq   : 'a * 'a queue -> unit
val frontq : 'a queue -> 'a
```

In the implementation, the `last` pointer is either points to the mutable `first` field or to the tail of the `qlist` stored there. This programming technique is widely used in C and it translates to ML with pointers very naturally. The implementation (see Fig. 14) is concise and readable. The counterpart using `ref` cells would be littered with exclamation marks [23].

Another application which benefits from pointers is the structure-sharing implementation of unification in a Prolog interpreter. The relevant `term` type is very simple in ML with pointers:

```
datatype term =
    Null
  | Var of ident * term
  | Con of ctor * term list
```

In Standard ML, the type would be sprinkled with `ref`s. As an additional benefit, the trail list (which contains the addresses of all variables that have been bound during unification) can be generalized to undo other temporary assignments to terms during execution, similar to the contents of the `first` field in the `queue` example.

```
datatype 'a qlist =
    QNil
  | QCons of 'a * 'a qlist;

type 'a queue =
   { first : 'a qlist,
     last  : 'a qlist ptr };

exception Queue_empty;

fun qnil QNil = true
  | qnil _    = false;

fun newq () =
  let val q = QNil
      val h = { first = QNil, last = &q)
  in
    h.last <- &(#first(h))
    h
  end;

fun mtq h = qnil (#first(h));

fun addq (h, x) =
  let val el = QCons (x, QNil)
  in
    #last(h) *= el;
    #last(h) <- &(#2(#QCons(el)))
  end;

fun frontq h =
  case #first(h) of
    QCons (x, rest) =>
      (#first(h) <- rest;
       if qnil rest
       then #last(h) <- &(#first(h))
       else ())
  | QNil =>
      raise Queue_Empty;
```

Figure 14: Implementation of mutable queues

4 Implementation issues

We have implemented the pointer extension in the Caml byte-code implementation of ML [18].

4.1 Implementation model

The implementation model suggested by the translation semantics in Sec. 1.4 introduces `ref` cells for everything whose address the program might take. This is to make sure every lvalue actually corresponds to a location. In practice however, ML implementations already provide the necessary locations, and it is thus possible to avoid most of the additional reference cells. Here are the alternatives for lvalues:

Variables already have locations within the local environment, a closure environment, or the global environment.

Record fields already have locations within the record.

Tuple fields already have locations within the tuple.

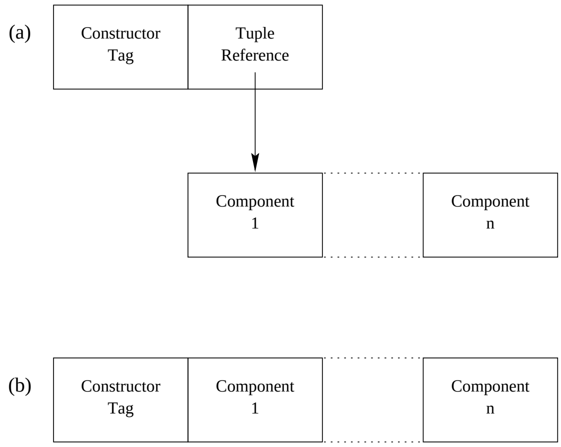

Figure 15: Constructed object representation—imagined and real

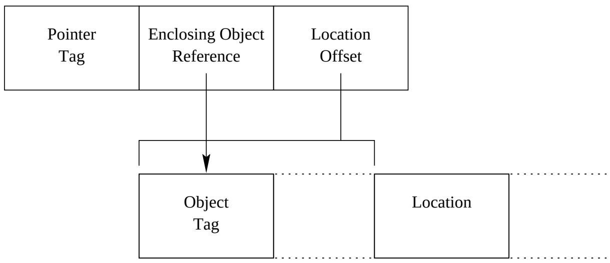

Figure 16: Pointer Representation

Constructor arguments have locations within the constructed object *unless the constructor argument is a tuple.* In that case, typical ML implementations (such as Caml) represent the tuple within the object in an unboxed form.

Figure 15 shows the difference between the model that the ML standard presents to the programmer **(a)** and the internal model used by the implementation **(b)**. Most ML implementations simply identify the constructed object with the tuple argument, and the constructor tag doubles as the (ignored) tuple tag.

Consequently, if the tuple argument of a constructor is addressed, the implementation needs to insert an indirection as in **(b)** via the translation described in Sec. 1.4.

4.2 Pointer representation

A pointer to a location also represents a reference to the enclosing object. Such "interior pointers" are problematic for precise garbage collectors [9, 8, 2]. Hence, our prototype implements pointers as objects with two fields: A reference to the enclosing object and an offset within the object to the location itself. Figure 16 shows the layout. This means that the garbage collector needs no modifications whatsoever to accomodate pointers.

4.3 New byte-code instructions

Caml's virtual machine is an efficient byte-code interpreter. It keeps parameters and `let`-bound variables on the stack, and propagates special intermediate result in an accumulator. Implementing pointers requires a number of pointer creation primitives—one for each surrounding object:

- `PTR` for pointers to `let`-bound locals.
- `ENVPTR` for pointers to `function`-bound locals.
- `PTRGLOBAL` for pointers to global variables.
- `PTRFIELD` for fixed-offset pointers to components of records or tuples.
- `PTRARRAYREFU` for variable-offset pointers to components of arrays.

With these in place, one implementation each for the `*` and `*=` operation suffices:

- `DEREF` for dereferencing a pointer.
- `SETPOINTER` for changing a pointer location.

The implementation of the primitives, as well as the compilation issues are all straightforward.

4.4 Local variables

Another issue with implementing the extension are pointers to `let`-bound and `fn`-bound variables—*local variables* for short—as pointed out in Sec. 1.2. The Caml engine stores these on a stack. Hence, local variables already have locations, but these may have a shorter lifespan than the extent of pointers referring to them. In purely stack-based implementations, inserting `ref` cells as in the formal translation is the only solution.

Things are easier in implementations which allow the transfer of activation frames to the heap, or which keep them on the heap in the first place. The typing rules in Sec. 1.3 already identify the locations whose environments need to be heap-allocated. Hence it is possible to insert any necessary environment stack-to-heap copy instructions at compile time.

Since a heap representation for the local environment is a natural requirement for languages with `call-with-current-continuation`, implementing the extension is possible without explicit boxing here. Scheme 48 [15], and RScheme are notable examples.

Standard ML/NJ [4, 3, 5] always represents the local environment on the heap, and therefore does not even need the escape information from the effect analysis. A simple, purely local, analysis would need to ensure that variables whose address the program takes do not end up in registers.

4.5 Polymorphic equality

Creating pointers separate as separate objects upon executing `&` operations solves the polymorphic equality problem trivially: Equality simply compares boxes introduced by the analysis by content, and pointers by address value.

4.6 Space leaks

The implementation approach described above has advantages over the naive application of the translation in Sec. 1.4 apart from implementation simplicity: Re-using already existing locations for constructing pointers avoids constructing additional `ref` cells. This means that normal access to addressed values remains unchanged—no extra indirection is necessary, and hence access incurs no performance penalty compared to access to non-addressed values.

However, re-using locations also has a downside: Taking a pointer of a value which is part of larger enclosing object also creates a reference to the enclosing object. As long as the pointer is live, the enclosing objects and everything accessible from it remains live as well. This may be particularly annoying in the case of pointers into the local environment, as it typically makes a whole chain of previous environments accessible. Thus, this implementation approach does not meet natural safe-for-space criterion. It is not clear whether such space leaks are relevant in practice. Also, it remains to be investigated if a variation on weak pointers could solve the problem.

4.7 Complications

Of course, not all ramifications of introducing pointers into an ML implementation are obvious from the semantics or even our prototype implementation, most of them having to do with optimization. Clearly, introducing pointers requires re-examination of other program analyses such as alias analysis [10], escape analysis [7], region inference [30] etc. Also, obtaining information about object immutability for optimizing automatic storage management is harder with pointers than it is with `ref` cells. Still, all of these things clearly remain doable via translation or via a more direct approach which utilizes the information from the effect analysis, or another, more precise method for inferring effect information.

5 Related work

A mechanism to take "the address of something" primordially exists in assembler. Higher-level programming language with such facilities include C [16] and Modula-2 [22]. We borrowed the operator symbols for the address and dereferencing operators from C. C and Modula-2 are among the few surviving languages which have an address operator; even Limbo, Lucent's intended successor to C [20], only supports references-as-boxes.

Smith and Volpano describe a novel typing framework for ML [26] which introduces mutable bindings into the language. Their proposal is largely orthogonal to ours, as using a mutable binding instead of a `ref` cell sometimes avoids typing problems. Their framework is also able to type an `&` operator. However, their work makes no effort to address the representation issues which are crucial to implementation. Smith and Volpano also describe a polymorphic type system for a dialect of C [27]. Due to the first-order nature of C and the absence of any execution-time safety guarantees, the problem is somewhat simpler than it is in ML.

Edinburgh LCF [11] also had a `letref` construct for mutable local variables. Albeit, it lacked first-class references entirely. Earlier versions of Caml and Caml-Light also provided mutable constructor arguments, albeit with different syntax from the one presented here [17].

A number of C compilers and debugging aids use a representation of pointers similar to ours to make their usage safe [25, 14, 12, 6]. Their representations typically include a bounds argument as well to support spatially safe pointer arithmetic. Making pointers completely safe in C is significantly harder than in ML because a pointer to a local variable has only limited extent, namely the lifetime of its activation record. Catching errors related to this requires a capability mechanism.

Our inference of boxing and escaping information is a simple form of monomorphic region inference [28, 30, 13]. Conceivably, a more precise region inference algorithm could improve the results of the analysis and reduce boxing.

6 Conclusion

We have explored the theoretical, practical, and technical implications of introducing a pointer mechanism into ML which allows taking the address of an object. Such an extension is possible, useful, and safe. The pointer mechanism allows for a more flexible programming style when dealing with mutable data, and alleviates maintenance problems which come with the use of `ref` cells. Not being able to take the address of an object seems to be a frequent complaint among programmers, especially those switching over from languages which allow this. Thus, pointers would also help acceptance of ML in the real world.

Acknowledgements We thank Xavier Leroy for his eminently practical as well as readable Objective Caml system, and for (unknowingly) helping with implementing the extension.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., second edition, 1996.

[2] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3:343–380, 1990.

[3] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Proc. Functional Programming Languages and Computer Architecture 1987*, volume 274 of *Lecture Notes in Computer Science*, pages 301–324, Portland, Oregon, September 1987. Springer-Verlag.

[5] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszynski and Martin Wirsing, editors, *Proc. Programming Language Implementation and Logic Programming '91*, pages 1–13, Passau, Germany, August 1991. Springer-Verlag. LNCS 528.

[6] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Fla, USA, June 1994. ACM Press.

[7] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In Luca Cardelli, editor, *Proc. 25th Annual ACM Symposium*

on Principles of Programming Languages, pages 25–37, San Diego, CA, USA, January 1998. ACM Press.

[8] Hans-J. Boehm. Space efficient conservative garbage collection. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 197–206, Albuquerque, New Mexico, June 1993.

[9] Hans-J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice & Experience*, 18(9):807–820, September 1988.

[10] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher order functional specifications. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, pages 157–168, San Francisco, CA, January 1990. ACM Press.

[11] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[12] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 125–138, Berkeley, CA, USA, January 1991. Usenix Association.

[13] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*, pages 303–310, Orlando, Florida, January 1991. ACM Press.

[14] Stephen Kaufer, Russell Lopez, and Sesha Pratap. Saber-C — an interpreter-based programming environment for the C language. In USENIX Association, editor, *USENIX Conference Proceedings, Summer, 1988. San Francisco*, pages 161–171, Berkeley, CA, USA, 1988. USENIX.

[15] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[16] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1988. 2nd edition.

[17] Xavier Leroy. *The Caml Light system release 0.73 Documentation and user's manual*. INRIA, release 0.73 edition. From `ftp://ftp.inria.fl:/INRIA/lang/caml-light/`.

[18] Xavier Leroy. *The Objective Caml system, documentation and user's guide*. INRIA, release 1.07 edition. From `ftp://ftp.inria.fl:/INRIA/lang/caml-light/`.

[19] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, San Diego, California, January 1988. ACM Press.

[20] Lucent Technologies. *Limbo Language Definition*, 1998. `http://www.lucent-inferno.com/Pages/Developers/Documentation/Limbo_Ref%20/langdef.htm`.

[21] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[22] International Standards Organization. ISO/IEC 10514-1 (Modula-2, base language), 1996.

[23] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996. 2nd edition.

[24] John Reppy. Standard ML basis library, 1997. `http://www.cs.bell-labs.com/~jhr/sml/basis/index.html`.

[25] Graham Ross. Integral C – A practical environment for C programming. In *Proceedings of the Symposium on Practical Software Development Environments*, pages 42–48. ACM, January 1987.

[26] Geoffrey Smith and Dennis Volpano. Polymorphic typing of variables and references. *ACM Transactions on Programming Languages and Systems*, 18(3):254–267, May 1996.

[27] Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 1998. To appear.

[28] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–272, July 1992.

[29] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[30] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, OG, January 1994. ACM Press.